

# Árvores

# **Aulas Anteriores...**

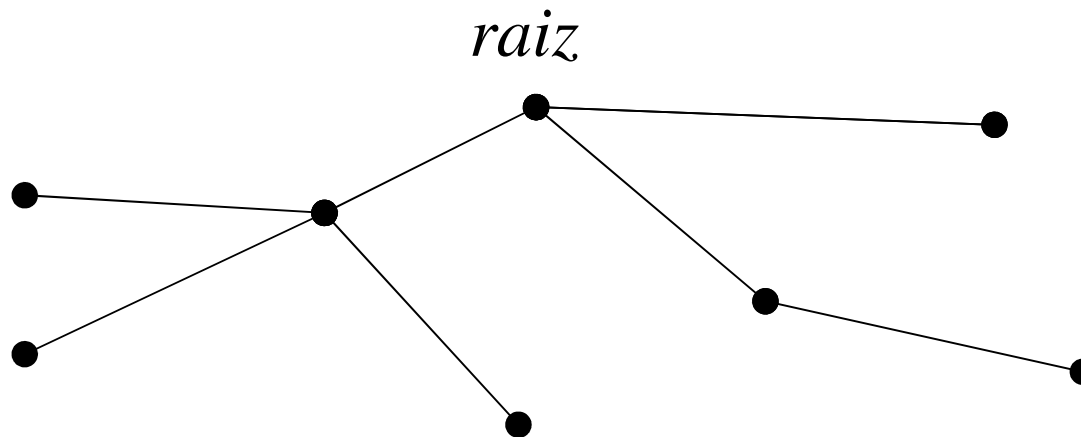
- Listas Lineares em alocação encadeada.
- Listas simplesmente encadeadas.
- Filas e Pilhas.
- Variações de listas encadeadas.
- Operações de inserção, remoção e busca nestas estruturas de dados.
- Análise das complexidades dos algoritmos apresentados.

# O que são árvores?

- Estruturas de dados não sequenciais (hierárquicas) de maior aplicação.
- Árvores binárias são mais comuns.
- Uso mais comum em recuperação de informação (árvores binárias de busca).

# Árvores

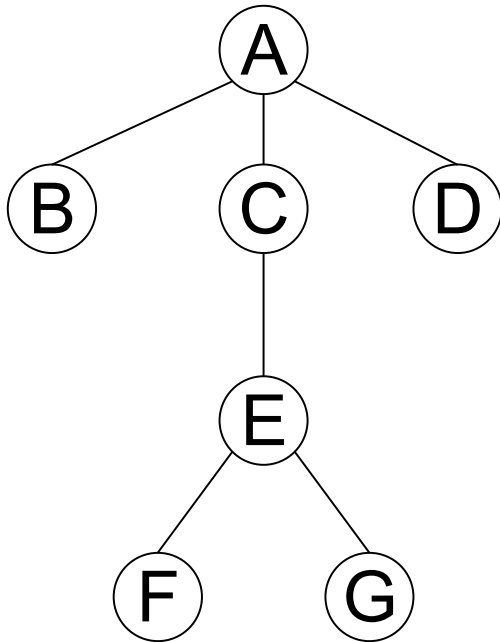
- Árvores são estruturas das mais usadas em computação.
- Árvores são usadas para representar hierarquias.
- Uma árvore pode ser entendida como um grafo acíclico conexo onde um dos vértices – chamado *raiz da árvore* – é diferenciado dos demais.



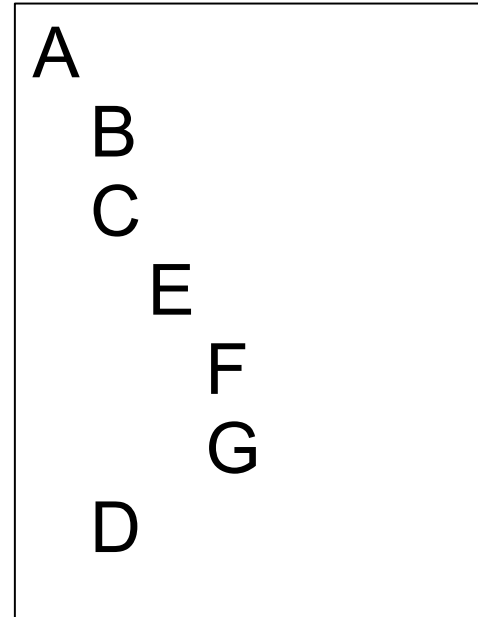
# Árvores

- Uma maneira mais útil de se definir árvores é a seguinte:
  - Uma árvore  $T$  é um conjunto finito de nós (ou vértices) tal que:
    - $T = \emptyset$ , isto é, uma árvore vazia.
    - Um nó raiz e um conjunto de árvores não vazias, chamadas de subárvores do nó raiz.
- É comum associar-se *rótulos* aos nós das árvores para que possamos nos referir a eles.
- Na prática, os nós são usados para guardar informações diversas.

# Árvores



Representação Gráfica

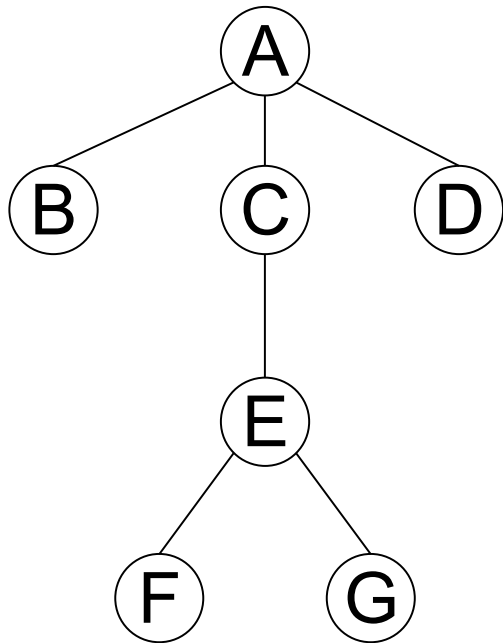


Representação Indentada

(A(B)(C(E(F)(G)))(D))

Representação com Parênteses

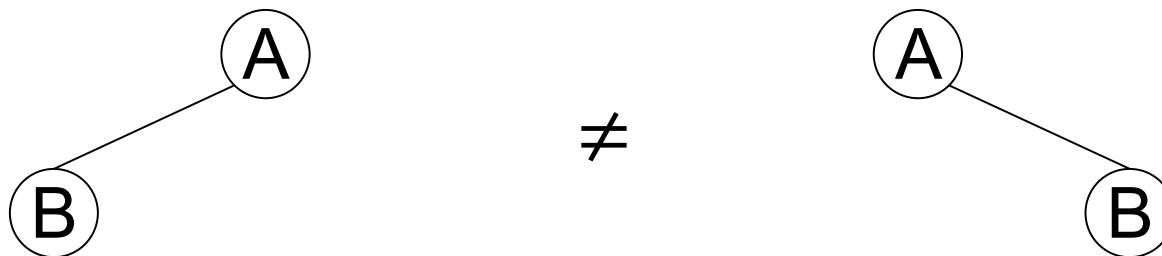
# Árvores – Nomenclatura



- “A” é o pai de “B”, “C” e “D”.
- “B”, “C” e “D” são filhos de “A”.
- “B”, “C” e “D” são irmãos.
- “A” é um ancestral de “G”.
- “G” é um descendente de “A”.
- “B”, “D”, “F” e “G” são nós folhas.
- “A”, “C” e “E” são nós internos.
- O grau do nó “A” é 3.
- O comprimento do caminho entre “C” e “G” é 2.
- O nível de “A” é 1 e o de “G” é 4.
- A altura da árvore é 4.

# Árvores Binárias

- Uma *árvore binária* é
  - Uma árvore vazia ou
  - Um nó *raiz* e duas subárvores binárias denominadas subárvore *direita* e subárvore *esquerda*
- Observe que uma árvore binária não é propriamente uma árvore já que os filhos de cada nó têm nomes (esquerdo e direito)



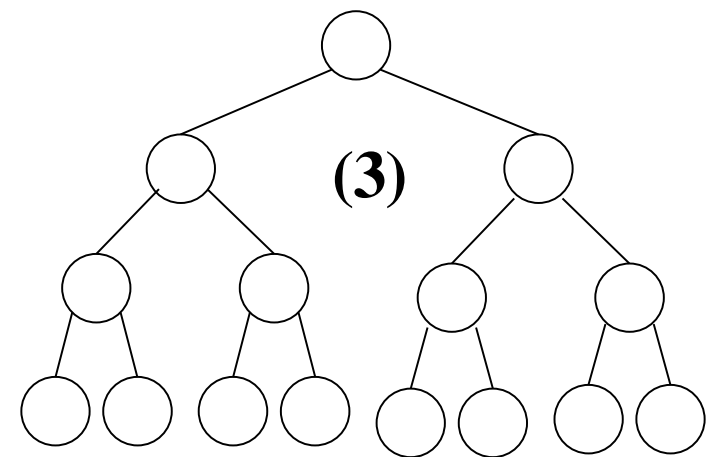
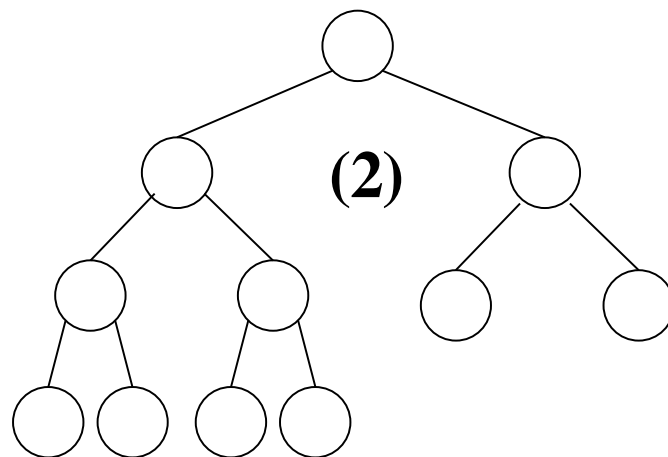
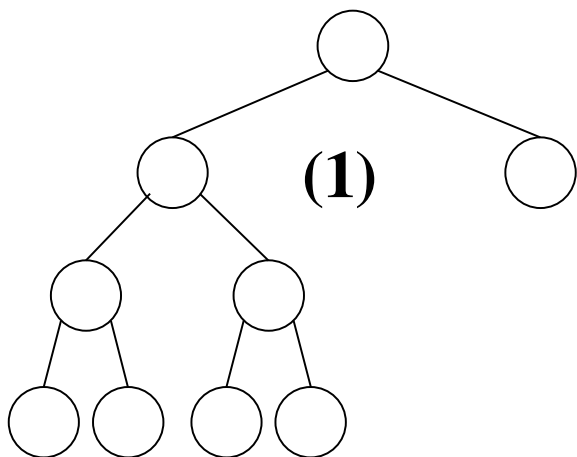


# Número de Subárvores Vazias

- Se uma árvore tem  $n > 0$  nós, então ela possui  $n+1$  subárvores vazias.
- Para verificar este “Lema”, observe que:
  - Uma árvore com um só nó tem 2 subárvores vazias
  - Sempre que “penduramos” um novo nó numa árvore, o número de nós cresce de 1 e o de subárvores vazias também cresce de 1

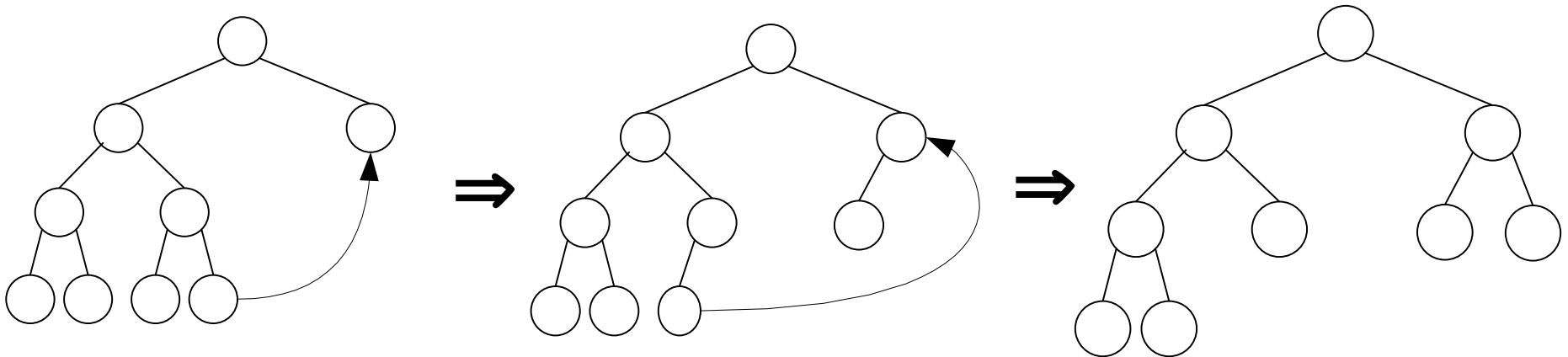
# Tipos Especiais de Árvores Binárias

- 1) Uma árvore binária é estritamente binária sse todos os seus nós têm 0 ou 2 filhos.
- 2) Uma árvore binária completa é aquela em que todas as subárvores vazias são filhas de nós do último ou penúltimo nível.
- 3) Uma árvore binária cheia é aquela em que todas as subárvores vazias são filhas de nós do último nível.



# Altura de Árvores Binárias

- Busca em árvores: a partir da raiz na direção de alguma de suas folhas.
- Árvores com a menor altura possível  $\Rightarrow$  busca mais eficiente.
- Se uma árvore  $T$  com  $n > 0$  nós é completa, então ela tem altura mínima. Uma árvore pode se tornar completa movendo-se folhas para níveis mais altos. Veja o exemplo abaixo:



# Altura de Árvores Binárias

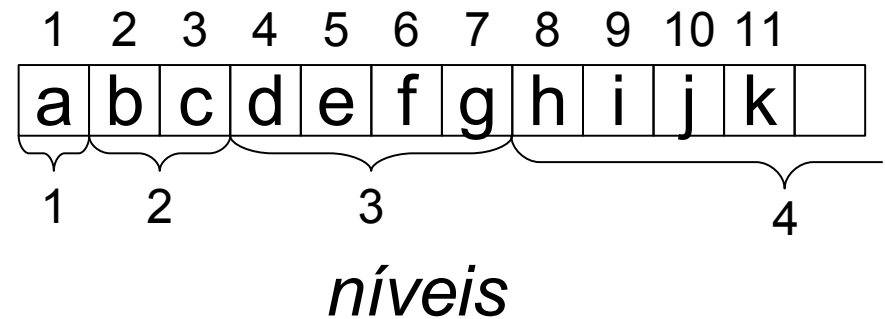
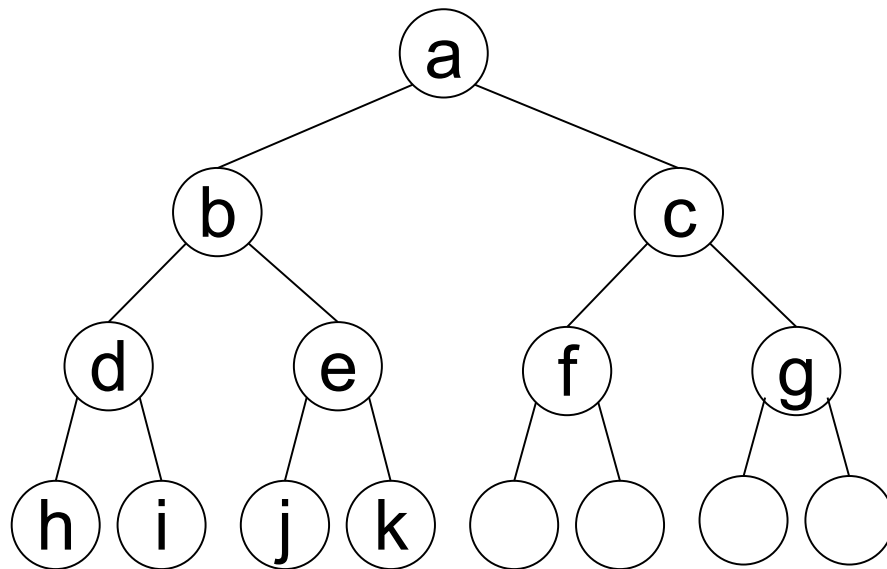
- A altura mínima de uma árvore binária com  $n > 0$  nós é:

$$h = 1 + \lfloor \log_2 n \rfloor$$

- Prova-se por indução. Seja  $T$  uma árvore completa de altura  $h$ 
  - Vale para o caso base ( $n=1$ )
  - Seja  $T'$  uma árvore cheia obtida a partir de  $T$  pela remoção de  $k$  folhas do último nível
    - Então  $T'$  tem  $n' = n - k$  nós
    - Como  $T'$  é uma árvore cheia,  
 $n' = 1 + 2 + \dots + 2^{m-1} = 2^m - 1$  ( $m = h-1$  é a altura da árvore) e  
 $h = 1 + \log_2 (n'+1)$
    - Sabemos que  $1 \leq k \leq n' + 1$  e portanto  
 $\log_2 (n'+1) = \lfloor \log_2 (n' + k) \rfloor = \lfloor \log_2 n \rfloor$

# Implementando AB com Vetores

- Assim como listas, árvores binárias podem ser implementadas utilizando-se vetores.
- A idéia é armazenar a árvore por níveis. Veja o exemplo abaixo:

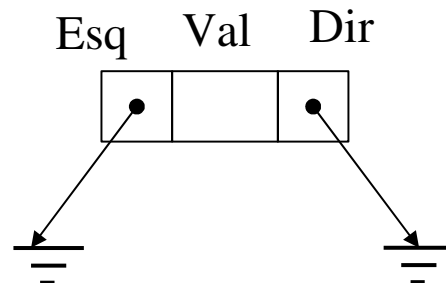


# Implementando AB com Vetores

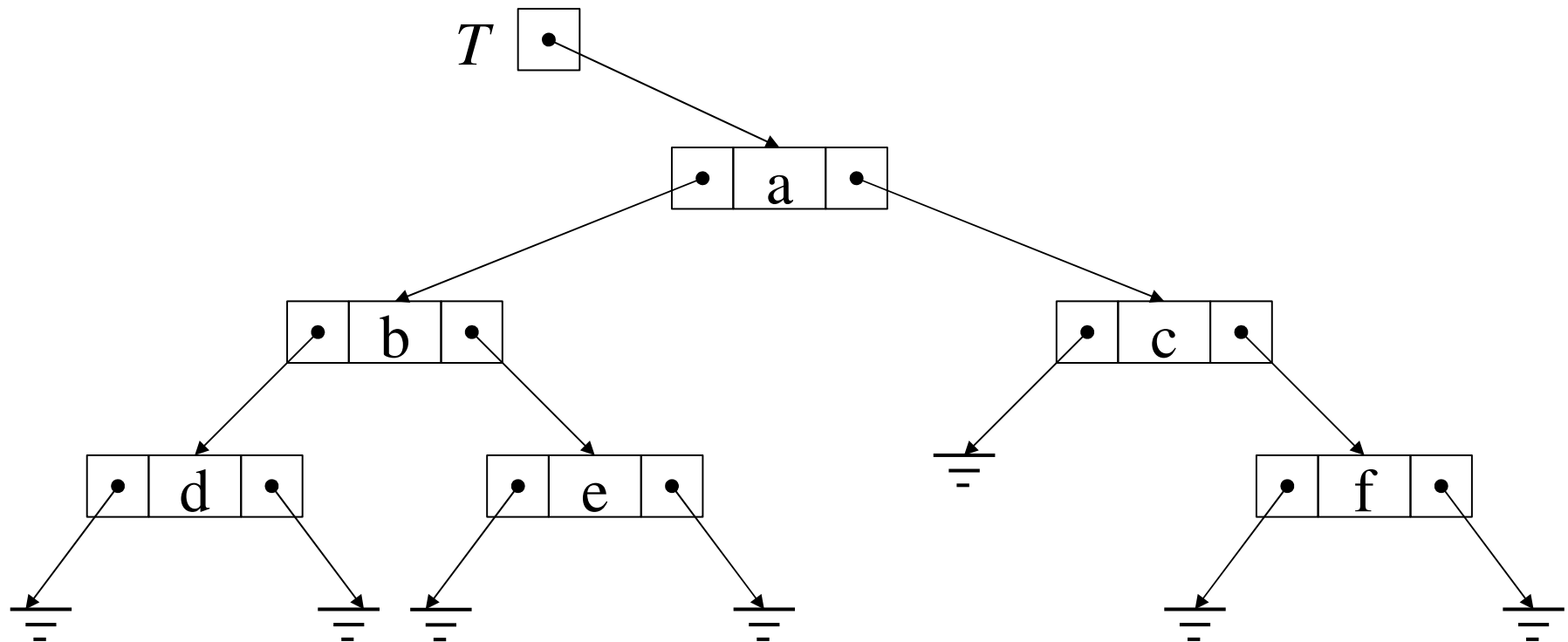
- Dado um nó armazenado no índice  $i$ :
  - O filho esquerdo de  $i$ :  $2i$
  - O filho direito de  $i$ :  $2i + 1$
  - O nó pai de  $i$ :  $i \text{ div } 2$
- Para armazenar uma árvore de altura  $h$  precisamos de um vetor de  $2^h - 1$ . Por quê?
- Nós correspondentes a subárvores vazias precisam ser marcados com um valor diferente de qualquer valor armazenado na árvore (*flag*).
- A cada índice computado é preciso se certificar que está dentro do intervalo permitido:
  - Ex.: O nó raiz é armazenado no índice 1 e o índice computado para o seu pai é 0

# Implementando AB com Ponteiros

- A implementação com vetores é simples porém tende a desperdiçar memória, e é pouco flexível quando se quer alterar a árvore (inserção e remoção de nós).
- Geralmente, as árvores são implementadas com ponteiros:
  - Cada nó  $X$  contém 3 campos:
    - $X.Val$  : valor armazenado no nó
    - $X.Esq$ : Ponteiro p/ árvore esquerda
    - $X.Dir$ : Ponteiro p/ árvore direita
  - Uma árvore é representada por um ponteiro para seu nó raiz.



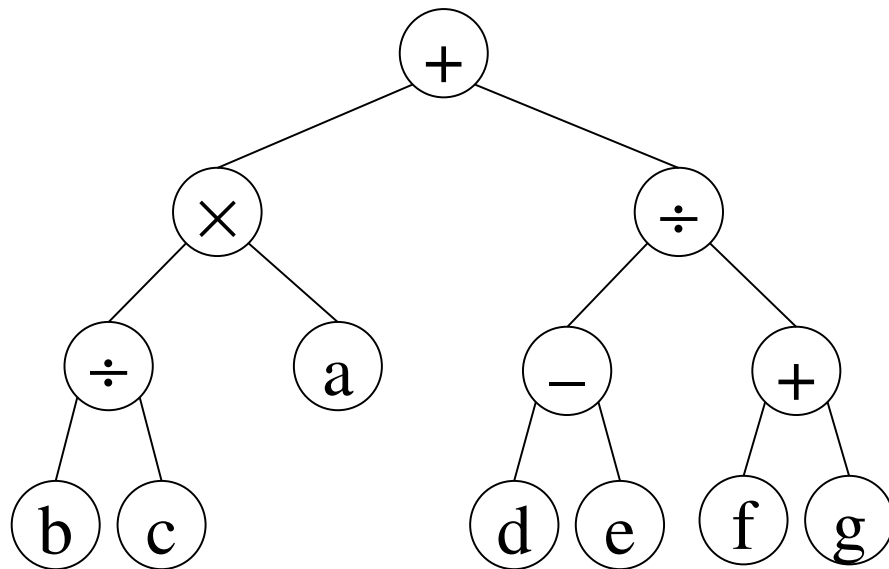
# Implementando AB com Ponteiros





# Aplicação: Expressões

- Uma aplicação bastante corriqueira de árvores binárias é na representação e processamento de expressões algébricas, booleanas, etc



$((b/c) * a) + ((d-e)/(f+g))$

# Avaliando uma Expressão

- Se uma expressão é codificada sob a forma de uma árvore, sua avaliação pode ser feita percorrendo os nós da árvore:

```
proc Avalia (Arvore T) {  
  se  $T^{\wedge}.Val$  é uma constante ou uma variável então  
    retornar o valor de  $T^{\wedge}.Val$   
  senão {  
     $operando1 \leftarrow Avalia(T^{\wedge}.Esq)$   
     $operando2 \leftarrow Avalia(T^{\wedge}.Dir)$   
    se  $T^{\wedge}.Val = "+"$  então  
      retornar  $operando1 + operando2$   
    senão se  $T^{\wedge}.Val = "-"$  então  
      retornar  $operando1 - operando2$   
    senão se  $T^{\wedge}.Val = "*"$  então  
      retornar  $operando1 * operando2$   
    senão se  $T^{\wedge}.Val = "/"$  então  
      retornar  $operando1 / operando2$   
  }  
}
```

# Percurso de Árvores Binárias

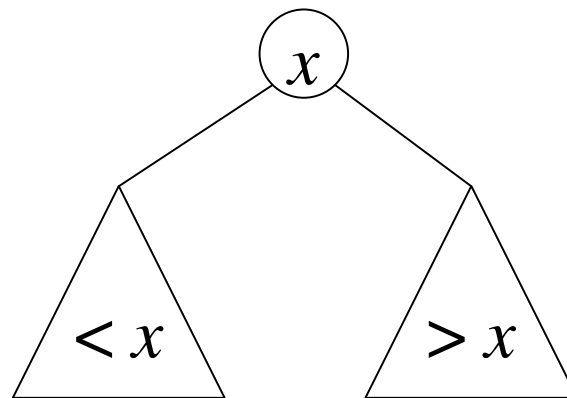
- Existem essencialmente 3 ordens “naturais” de se percorrer os nós de uma árvore:
  - Pré-ordem: raiz, esquerda, direita
  - Pós-ordem: esquerda, direita, raiz
  - In-ordem: esquerda, raiz, direita
- Por exemplo:
  - percorrendo uma árvore que representa uma expressão em in-ordem, obtém-se a expressão em sua forma usual (infixa);
  - um percurso em pós-ordem produz a ordem usada em calculadoras “HP”;
  - um percurso em pré-ordem retorna a expressão em forma infixada, como usado em LISP

# Dicionários

- A operação de busca é fundamental em diversos contextos da computação.
- Por exemplo, um *dicionário* é uma estrutura de dados que reúne uma coleção de chaves sobre a qual são definidas as seguintes operações :
  - *Inserir* ( $x, T$ ) : inserir chave  $x$  no dicionário  $T$
  - *Remover* ( $x, T$ ) : remover chave  $x$  do dicionário  $T$
  - *Buscar* ( $x, T$ ) : verdadeiro apenas se  $x$  pertence a  $T$
- Outras operações são comuns em alguns casos:
  - Encontrar chave pertencente a  $T$  que sucede ou precede  $x$
  - Listar todas as chaves entre  $x_1$  e  $x_2$

# Árvores Binárias de Busca

- Uma maneira simples e popular de implementar dicionários é uma estrutura de dados conhecida como árvore binária de busca.
- Numa árvore binária de busca, todos os nós na subárvore à esquerda de um nó contendo uma chave  $x$  são menores que  $x$  e todos os nós da subárvore à direita são maiores que  $x$ .

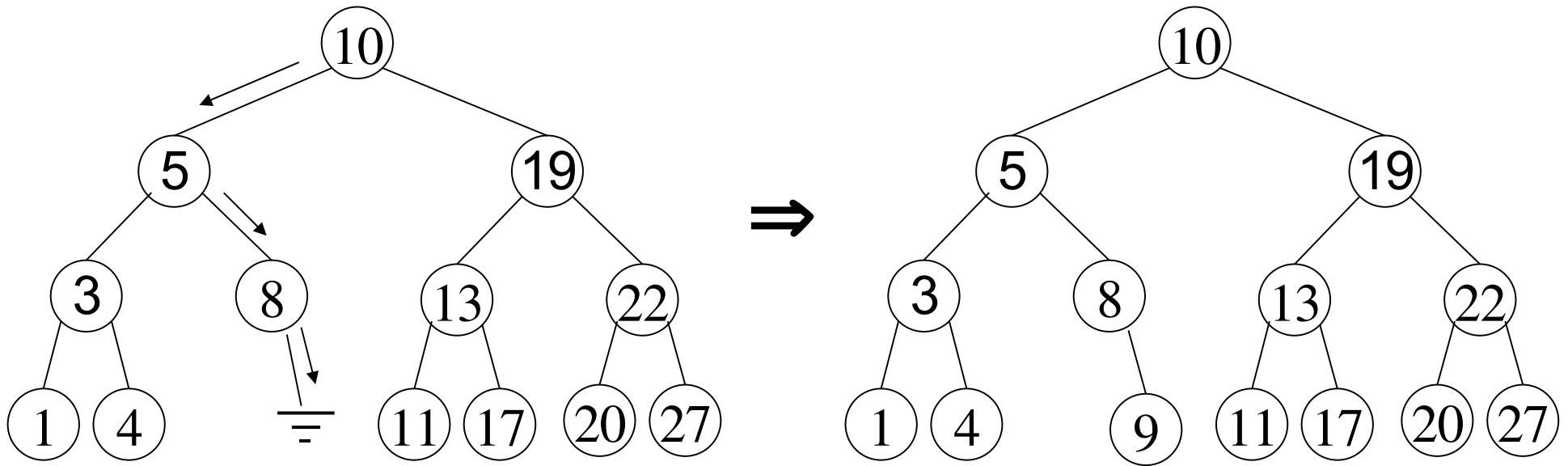


# Busca e Inserção em ABB

```
proc Buscar (Chave x, Árvore T) {  
    se  $T = \text{Nulo}$  então retornar falso  
    se  $x = T^{\wedge}.Val$  então retornar verdadeiro  
    se  $x < T^{\wedge}.Val$  então retornar Buscar ( $x$ ,  $T^{\wedge}.Esq$ )  
    retornar Buscar ( $x$ ,  $T^{\wedge}.Dir$ )  
}  
  
proc Inserir (Chave x, var Árvore T) {  
    se  $T = \text{Nulo}$  então {  
         $T \leftarrow \text{Alocar}(\text{NoArvore})$   
         $T^{\wedge}.Val \leftarrow x$ ,  $T^{\wedge}.Esq \leftarrow \text{Nulo}$ ,  $T^{\wedge}.Dir \leftarrow \text{Nulo}$   
    }  
    senão {  
        se  $x < T^{\wedge}.Val$  então Inserir ( $x$ ,  $T^{\wedge}.Esq$ )  
        se  $x > T^{\wedge}.Val$  então Inserir ( $x$ ,  $T^{\wedge}.Dir$ )  
    }  
}
```

# Inserção em ABB

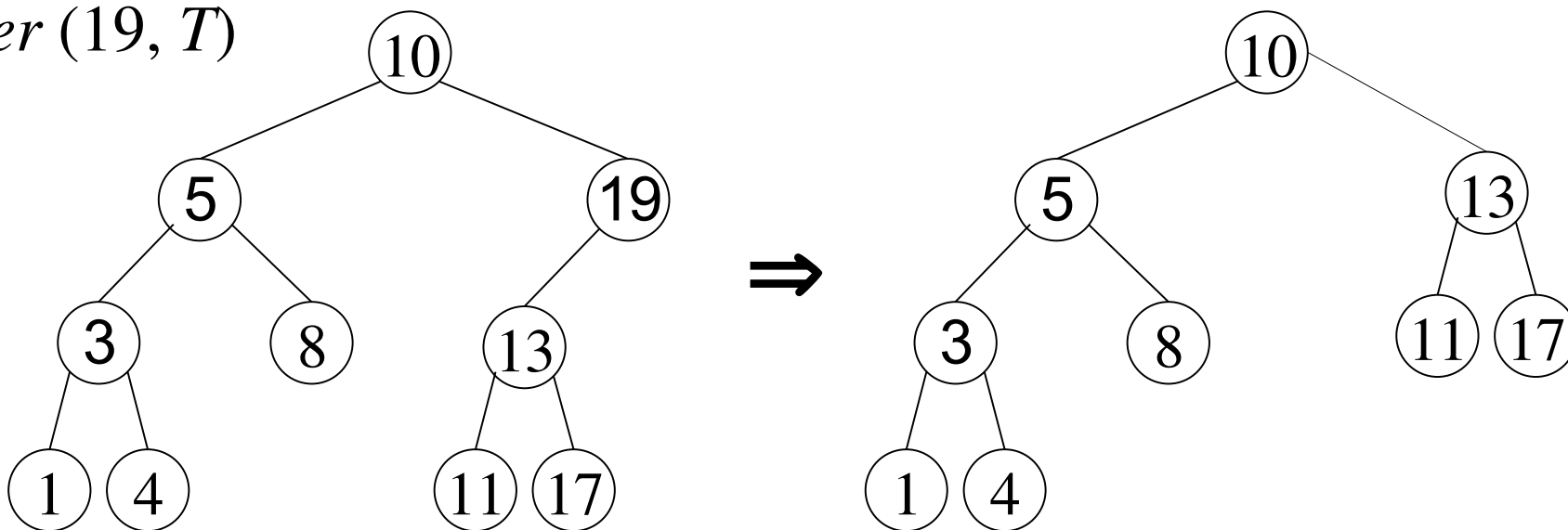
*Inserir (9, T)*



# Remoção em ABB

- Para remover uma chave  $x$  de uma árvore  $T$  temos que distinguir os seguintes casos
  - $x$  está numa folha de  $T$ : neste caso, a folha pode ser simplesmente removida
  - $x$  está num nó que tem sua subárvore esquerda ou direita vazia: neste caso o nó é removido substituído pela subárvore não nula

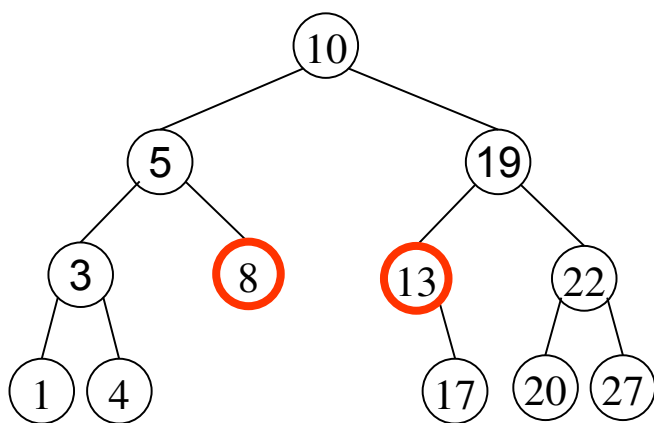
*Remover (19,  $T$ )*



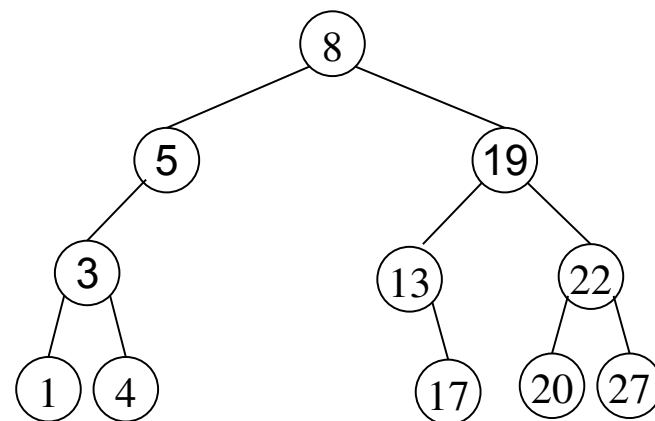
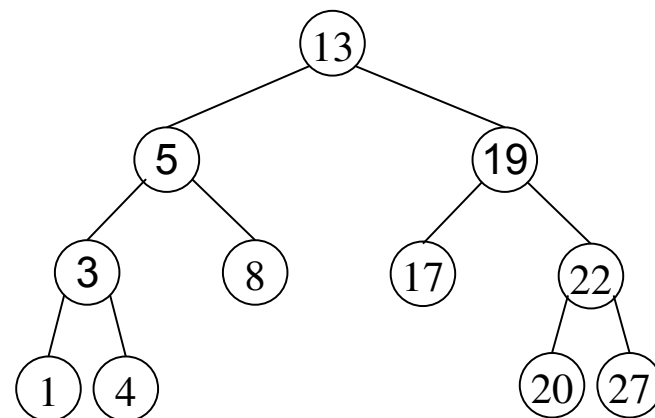


# Remoção em ABB

- Se  $x$  está num nó em que ambas subárvores são não nulas, é preciso encontrar uma chave  $y$  que a possa substituir. Há duas chaves candidatas naturais:
  - A menor das chaves maiores que  $x$  ou
  - A maior das chaves menores que  $x$



*Remover* (10,  $T$ )



# Remoção em ABB

```
proc RemoverMenor (var Árvore T) {  
    se  $T^{\wedge}.Esq = Nulo$  então {  
         $tmp \leftarrow T$   
         $y \leftarrow T^{\wedge}.Val$   
         $T \leftarrow T^{\wedge}.Dir$   
        Liberar (tmp)  
        retornar y  
    }  
    senão  
        retornar RemoverMenor ( $T^{\wedge}.Esq$ )  
}
```

**Mostrar graficamente.**

# Remoção em ABB

```
proc Remover (Chave  $x$ , Árvore  $T$ ) {  
    se  $T \neq \text{Nulo}$  então  
        se  $x < T^{\wedge}.val$  então Remover ( $x$ ,  $T^{\wedge}.Esq$ )  
        senão se  $x > T^{\wedge}.val$  então Remover ( $x$ ,  $T^{\wedge}.Dir$ )  
        senão  
            se  $T^{\wedge}.Esq = \text{Nulo}$  então {  
                 $tmp \leftarrow T$   
                 $T \leftarrow T^{\wedge}.Dir$   
                Liberar ( $tmp$ )  
            }  
            senão se  $T^{\wedge}.Dir = \text{Nulo}$  então {  
                 $tmp \leftarrow T$   
                 $T \leftarrow T^{\wedge}.Esq$   
                Liberar ( $tmp$ )  
            }  
            senão  $T^{\wedge}.Val \leftarrow \text{RemoverMenor} (T^{\wedge}.Dir)$   
}
```

# Complexidade de ABB

- A busca em uma árvore binária tem complexidade  $O(h)$ .
- A altura de uma árvore é,
  - no pior caso,  $n$
  - no melhor caso,  $\lfloor \log_2 n \rfloor + 1$  (árvore completa)
- Inserção e remoção também têm complexidade de pior caso  $O(h)$ , e portanto, a inserção ou a remoção de  $n$  chaves toma tempo:
  - $O(n^2)$  no pior caso ou
  - $O(n \log n)$  se pudermos garantir que árvore tem altura logarítmica

# Árvores de Busca de Altura Ótima

- É fácil ver que podemos garantir uma árvore de altura ótima para uma coleção de chaves se toda vez que temos que escolher uma chave para inserir, optamos pela mediana:

```
proc InserirTodos (i, n, A [i .. i+n−1], var Árvore T) {  
    se n = 1 então Inserir (A [i], T)  
    senão {  
        j ← Mediana (i, n, A)  
        trocar A[i] com A [j]  
        m ← Particao (i, n, A)  
        Inserir (A [i+m], T)  
        InserirTodos (i, m, A, T^.Esq)  
        InserirTodos (i+m+1, n−m−1, A, T^.Dir)  
    }  
}
```

# Árvores Balanceadas

- Vimos que árvores completas garantem buscas utilizando, no máximo,  $\lfloor \log_2 n \rfloor + 1$  comparações.
- ABB, se construídas por inserção aleatória de elementos podem ter altura logarítmica (em  $n$ ). Entretanto, isso nem sempre acontece, pois depende da ordem de inserção.
- A idéia, então, é modificar os algoritmos de inserção e remoção de forma a assegurar que a árvore resultante tenha sempre de altura logarítmica.
- A variante mais conhecida é a AVL.

# Árvores AVL

- Reorganizar a árvore quando ela deixa de ser completa (devido a uma inserção ou remoção por exemplo) pode ser muito custoso (até  $n$  operações).
- Uma idéia é estabelecer um critério mais fraco que, não obstante, garanta altura logarítmica.
- O critério sugerido por Adelson-Velskii e Landis é o de garantir a seguinte invariante:
  - Para cada nó da árvore, a altura de sua subárvore esquerda e de sua subárvore direita diferem de, no máximo, 1.
- Para manter essa invariante depois de alguma inserção ou remoção que desbalanceie a árvore, utiliza-se operações de custo  $O(1)$  chamadas *rotações*.

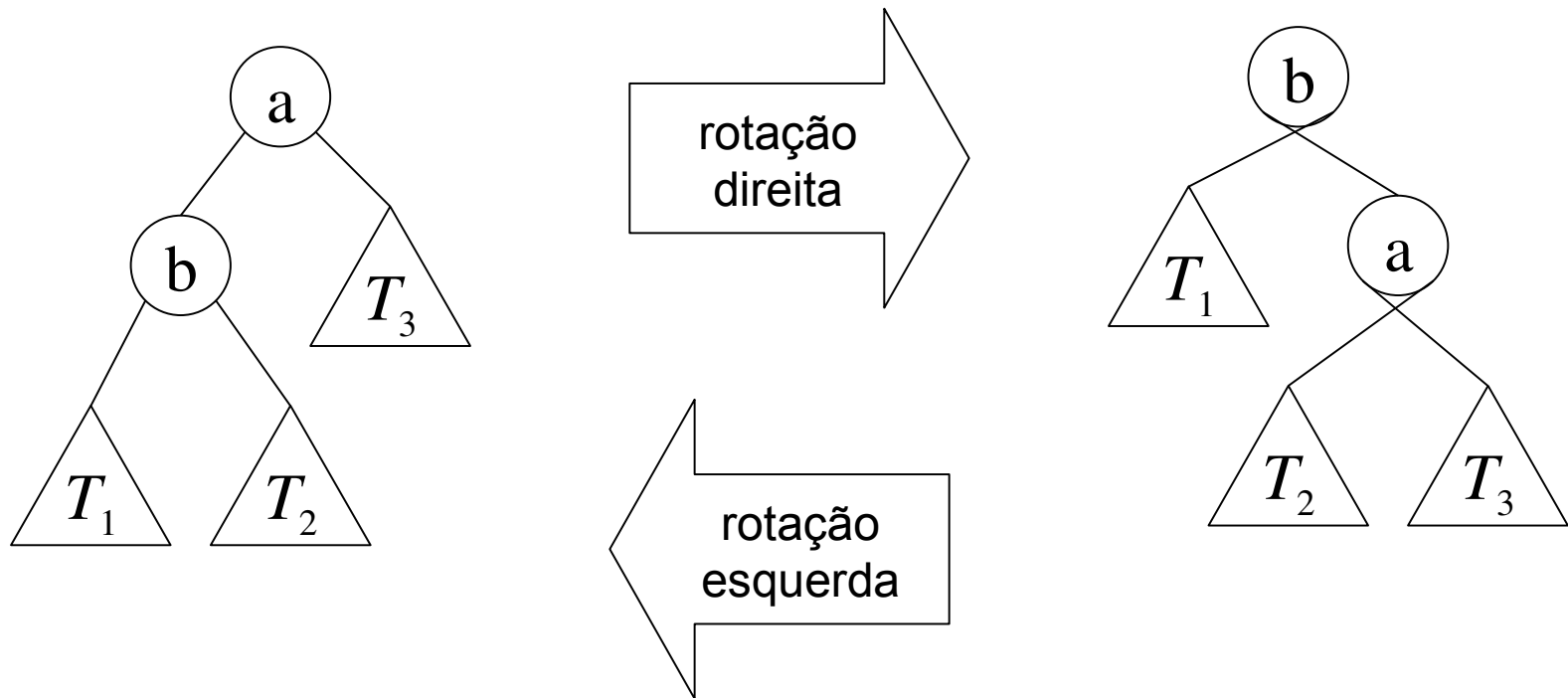
# Árvores AVL

- Uma árvore AVL tem altura logarítmica?
  - Seja  $N(h)$  o número mínimo de nós de uma árvore AVL de altura  $h$ .
  - Claramente,  $N(1) = 1$  e  $N(2) = 2$
  - Em geral,  $N(h) = N(h-1) + N(h-2) + 1$
  - Essa recorrência é semelhante à recorrência obtida para a série de Fibonacci.
  - Resolução da recorrência de Fibonacci:

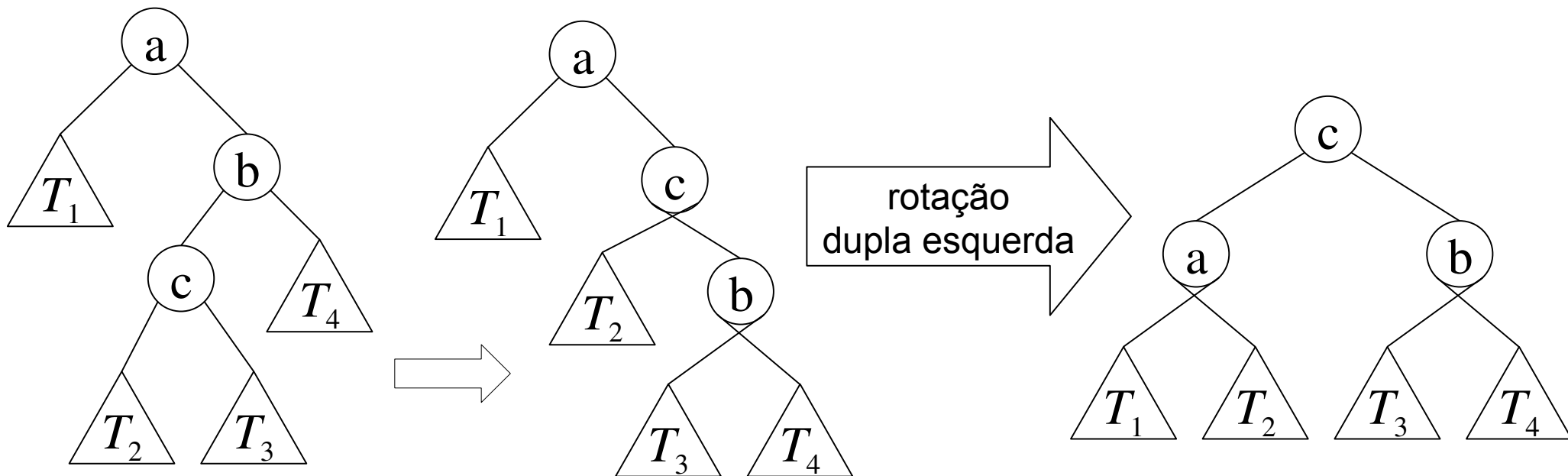
$$F_h = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^h - \left( \frac{1 - \sqrt{5}}{2} \right)^h \right]$$



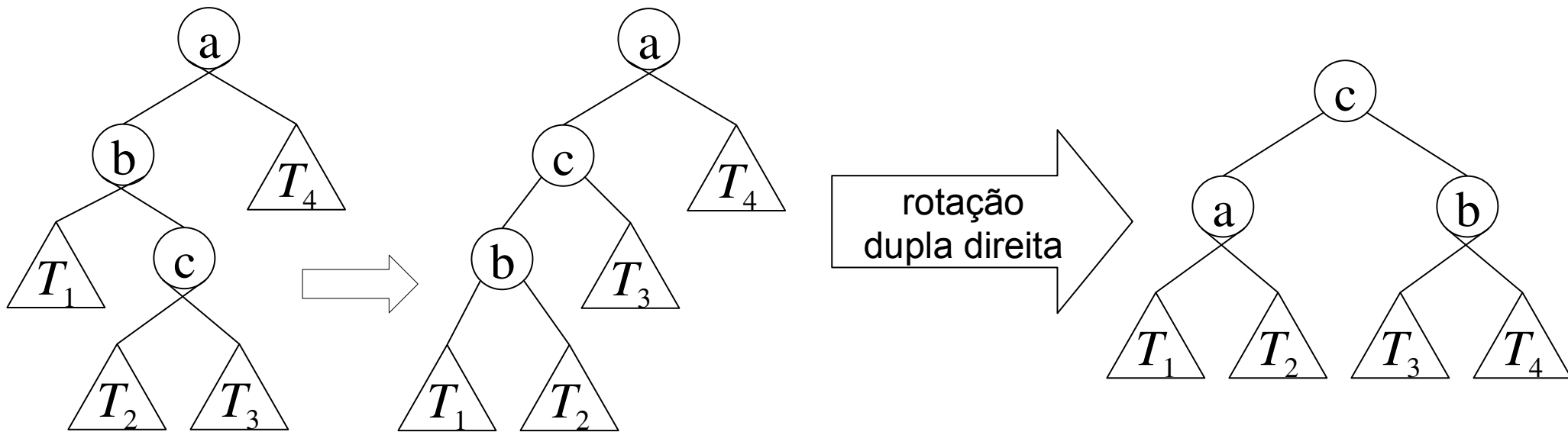
# Rotações em AVL



# Rotações em AVL



# Rotações em AVL



# Inserção em árvores AVL

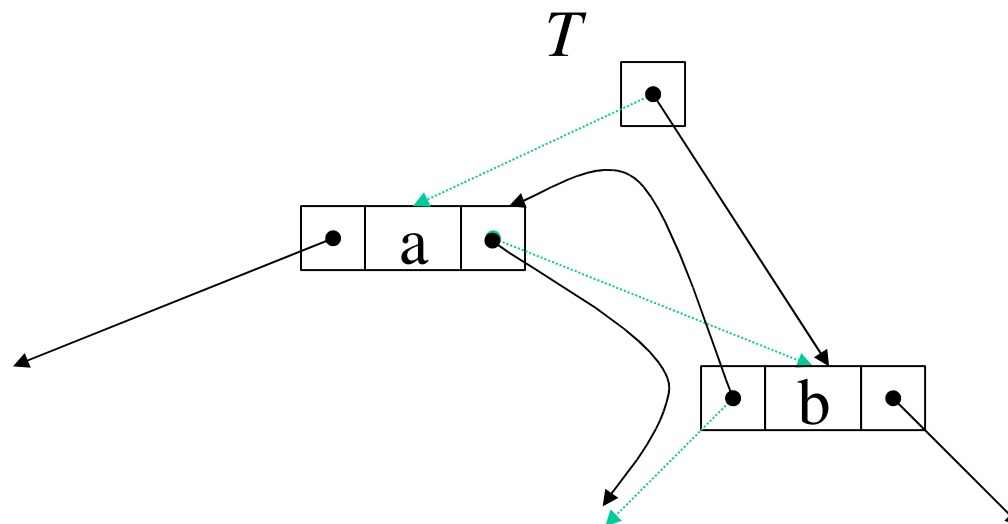
- Precisamos manter em cada nó um campo extra chamado *alt* que vai registrar a altura da árvore ali enraizada.
  - Na verdade, apenas a diferença de altura entre a subárvore esquerda e direita precisa ser mantida (2 bits).
- Vamos precisar das seguintes rotinas para acessar e atualizar as alturas das árvores:

```
proc Altura (Arvore T) {  
    se  $T = \text{Nulo}$  então retornar 0 senão retornar  $T^{\wedge}.Alt$   
}
```

```
proc AtualizaAltura (Arvore T) {  
    se  $T \neq \text{Nulo}$  então  
         $T^{\wedge}.Alt \leftarrow \max (\text{Altura} (T^{\wedge}.Esq), \text{Altura} (T^{\wedge}.Dir)) + 1$   
}
```

# Inserção em árvores AVL

```
proc RotacaoEsquerda (var Arvore T) {  
     $T \leftarrow T^{\wedge}.Dir$ ,  $T^{\wedge}.Dir \leftarrow T^{\wedge}.Dir^{\wedge}.Esq$ ,  $T^{\wedge}.Dir^{\wedge}.Esq \leftarrow T$   
    AtualizaAltura ( $T^{\wedge}.Esq$ )  
    AtualizaAltura (T)  
}
```



# Inserção em árvores AVL

```
proc RotacaoDireita (var Arvore T) {  
    T, T^.Esq, T^.Esq.Dir    T^.Esq, T^.Esq^.Dir, T  
    AtualizaAltura (T^.Dir)  
    AtualizaAltura (T)  
}  
  
proc RotacaoDuplaEsquerda (var Arvore T) {  
    RotacaoDireita (T^.Dir)  
    RotacaoEsquerda (T)  
}  
  
proc RotacaoDuplaDireita (var Arvore T) {  
    RotacaoEsquerda (T^.Esq)  
    RotacaoDireita (T)  
}
```

# Inserção em árvores AVL

```
proc InserirAVL (Chave x, var Arvore T) {  
    se  $T = \text{Nulo}$  então {  
         $T \leftarrow \text{Alocar}(\text{NoArvore})$   
         $T^{\wedge}.Val \leftarrow x, T^{\wedge}.Esq \leftarrow \text{Nulo}, T^{\wedge}.Dir \leftarrow \text{Nulo}, T^{\wedge}.Alt \leftarrow 1$   
    } senão {  
        se  $x < T^{\wedge}.Val$  então {  
            InserirAVL ( $x, T^{\wedge}.Esq$ )  
            se  $\text{Altura}(T^{\wedge}.Esq) - \text{Altura}(T^{\wedge}.Dir) = 2$  então  
                se  $x < T^{\wedge}.Esq^{\wedge}.Val$  então RotacaoDireita ( $T$ )  
                senão RotacaoDuplaDireita ( $T$ )  
        } senão {  
            InserirAVL ( $x, T^{\wedge}.Dir$ )  
            se  $\text{Altura}(T^{\wedge}.Dir) - \text{Altura}(T^{\wedge}.Esq) = 2$  então  
                se  $x > T^{\wedge}.Dir^{\wedge}.Val$  então RotacaoEsquerda ( $T$ )  
                senão RotacaoDuplaEsquerda ( $T$ )  
        }  
        AtualizaAltura ( $T$ )  
    }  
}
```

# Análise do Algoritmo de Inserção em AVLs

- Pode-se ver que apenas uma rotação (dupla ou simples) no máximo é necessária ( $O(1)$ ).
- A atualização do campo altura ( $O(1)$ ) pode ter de ser feita mais do que uma vez.
  - Na verdade, tantas vezes quantos forem os nós no caminho até a folha inserida.
  - No total,  $O(\log n)$ .
- No mais, o algoritmo é idêntico ao da inserção em árvores binárias de busca, e portanto a complexidade é  $O(\log n)$ . A vantagem está na melhoria do tempo de busca.



# Remoção em Árvores AVL

- Segue as mesmas linhas do algoritmo de inserção:
  - Faz-se a remoção do nó como uma árvore de busca comum.
  - Analisa-se a informação de balanceamento aplicando a rotação apropriada se for necessário.
- Diferentemente da inserção, pode ser necessário realizar mais do que uma rotação:
  - Na verdade, até  $\log n$  rotações.
  - Não afeta a complexidade do algoritmo de remoção que continua  $O(\log n)$ .

# Remoção em Árvores AVL

